Feature Flags

Introduction

In the Play+ ecosystem, we believe shipping code should be empowering, not risky. This helper is based on the concept of decoupling deployment from release , a modern development practice that allows us to merge and deploy features to production safely, while precisely controlling their visibility.

Feature flags (or toggles) are essential for mitigating risk, enabling A/B testing, and personalizing user experiences at scale. This aligns directly with our Adaptive pillar, allowing the application to change dynamically based on user context or business needs. It also promotes an Intuitive development workflow by simplifying a complex topic and makes the product more Engaging by allowing for controlled experiments with new features.

Package Info

The playfeature helper is provided through the @playplus/features package, which is included by default in the Golden Path.

Description	Package / Path
Golden Path (Recommended)	Pre-installed (/system/play.feature.ts)
Uplift Path	npm install @playplus/features

Folder Reference

The feature flag helper and its configuration follow our standardized folder structure for core system logic.

File / Directory	Purpose & Guidelines
system/play.feature.ts	The core feature flag service. It abstracts the third-party provider and provides a consistent API.

File / Directory	Purpose & Guidelines
config/play.feature.config.json	User-overridable configuration for the flag provider, client keys, and default values for offline development.

Helper - Pillars Alignment

The playfeature helper is a key enabler of our core design pillars.

Pillar	How This Helper Aligns
Adaptive	Primary Pillar : Allows the application's UI and behavior to adapt in real-time based on user, region, or experimental group.
Intuitive	Abstracts the complexity of third-party SDKs into a simple, synchronous API, making it trivial for developers to use flags.
Engaging	Empowers teams to safely A/B test new, delightful features and interactions, gathering data to create more engaging experiences.

Helper Overview

The playfeature helper is your application's runtime control panel. It provides a clean, synchronous, and framework-agnostic interface to your feature flagging provider (e.g., LaunchDarkly, Optimizely, Flagsmith). Its purpose is to abstract the plumbing of flag management, so developers can focus on building features, not on SDK integration.

Behind the scenes, the helper automates the entire process:

- Initialization : On app startup, it connects to your flag provider using the configured client key.
- User Identification : It automatically identifies the current user to fetch targeted flags.
- Real-time Updates : It maintains a live connection, so any change made in your provider's dashboard is reflected in the app in real-time, without a page refresh.

• Offline Graceful Degradation : If the provider is unreachable, it falls back to default values defined in the configuration.

This means a developer can simply ask playfeature.isEnabled('my-flag') and trust that the system is handling all the complex state management in the background.

Config Options

Global configuration is managed in config/play.feature.config.json .

Config Variable	Default Value	Description	Recommended Value
provider	"local"	The name of your feature flag provider (e.g., launchdarkly, flagsmith, optimizely). local uses bootstrap values.	Your provider's name.
clientKey	null	The client-side ID or key for your feature flag project.	Your provider's client key.
bootstrap	{}	An object of default flag values to use during initial load or for offline development and testing.	Define key flags for testing.

Example play.feature.config.json:

Helper Methods

The helper provides a simple, consistent API for accessing flag values.

Method Name	What It Does	Method Signature
isEnabled	Checks if a boolean feature flag is enabled. Returns defaultValue if the flag doesn't exist.	isEnabled(key: string, defaultValue?: boolean): boolean
getVariant	Gets the string value for a multivariate or experiment flag.	getVariant(key: string, defaultVariant?: string): string
onUpdate	Subscribes to all flag changes from the provider, allowing the UI to react in real-time.	onUpdate(callback: () => void): void
offUpdate	Unsubscribes from flag changes to prevent memory leaks.	offUpdate(callback: () => void): void

Angular Integration

PlayFeatureService

Angular service wrapper that integrates with Play+ logging and provides component-specific feature flag utilities.

PlayFeatureGuard

Route protection guard for feature flag-based access control.

FeatureFlagDirective

Template conditional rendering based on feature flags.

FeatureFlagPipe

Template flag checks and variant access.

Usage Examples

React: The useFeature Hook

For a seamless experience in React, the starter kit provides a useFeature hook that automatically subscribes to live updates.

Angular: The PlayFeatureGuard for Routes

For Angular, a CanActivate guard is the perfect way to toggle access to entire feature routes.

Service-based Flag Checks

** Recommended: Use PlayFeatureService**

Reactive Flag Observables

** Recommended: Use reactive observables**

Additional Info

Why We Created This Helper

Without a centralized helper, each developer would need to:

- Integrate and learn a complex, third-party feature flag SDK.
- Manually handle user identification for targeted rollouts.
- Build their own logic for real-time updates and subscriptions.
- Write boilerplate code to handle cases where the flag provider is offline.

This is inefficient and error-prone. The playfeature helper abstracts all of this into a single, reliable system. It provides a consistent, tested interface so developers can add or check a feature flag in a single line of code, trusting that the underlying complexity is managed for them.

Best Practices

- Keep Flags Short-Lived : Flags are temporary. Create a process for cleaning them up after a feature is fully rolled out to avoid technical debt.
- Use Descriptive Names : A good name like feat-newDashboard-rollout-2025-q3 is self-documenting. It tells you the feature, its purpose, and its expected lifespan.

- Abstract Flag Checks : Don't sprinkle playfeature.isEnabled() calls directly in JSX. Encapsulate the logic in a variable, hook (useFeature), or function for better readability and maintenance.
- Always Provide a Default Value : This ensures your application behaves predictably if the flagging service is down or a flag is deleted.

Flag Naming Convention

** Good Names:**

- feat-new-dashboard-2025-q3
- enable-beta-analytics
- feat-secure-input-validation
- feat-performance-monitoring

** Bad Names:**

- newDashboard (no prefix)
- beta (too vague)
- flag1 (not descriptive)
- temp (temporary flags become permanent)

Default Values

** Always provide safe defaults:**

** Avoid implicit defaults:**

Flag Abstraction

** Abstract flag checks:**

** Don't sprinkle flag checks everywhere:**

Flag Lifecycle Management

** Plan flag cleanup:**

Forbidden Patterns

1. Direct playfeature Access

** Never access playfeature directly:**

2. Magic String Flag Names

** Don't use magic strings:**

3. Inline Flag Checks in Templates

** Don't put complex flag logic in templates:**

4. Flag Checks in Services

** Don't check flags in business logic services:**

Required Patterns

1. Use PlayFeatureService

** Always use the Angular service:**

2. Provide Default Values

** Always provide explicit defaults:**

3. Use Reactive Patterns

** Use observables for reactive UI:**

4. Abstract Flag Logic

** Encapsulate flag checks:**

Configuration Management

Environment-specific Configuration

User Context

Testing

Testing with Different Flag States

** Test with different flag states:**

Testing Checklist

- Test with flag enabled
- Test with flag disabled
- Test with flag not defined (uses default)
- Test flag changes during runtime
- Test route protection with guard
- Test directive conditional rendering
- Test pipe usage in templates
- Test user context changes
- Test offline/fallback behavior

Monitoring & Observability

Flag Usage Tracking

Flag Performance Monitoring

Enforcement

ESLint Rules

The following ESLint rules enforce Play+ feature flag patterns:

- No direct playfeature access - Forces use of PlayFeatureService
- No magic string flag names - Requires flags to be defined in config
- No inline flag logic - Encourages abstraction

Code Review Checklist

- Uses PlayFeatureService instead of direct playfeature access
- Provides explicit default values
- Uses reactive patterns where appropriate
- Abstracts flag logic from templates
- Follows naming conventions

- Includes cleanup plan for new flags
- Tests cover different flag states

Migration Guide

From Direct playfeature Access

Before:

After:

From Manual Flag Management

Before:

After:

Troubleshooting

Common Issues

- Flag not working - Check if flag is defined in config
- Service not initialized - Ensure PlayFeatureService.initialize() is called
- Route protection not working - Verify guard is properly configured
- Template not updating - Use reactive observables instead of one-time checks

Debug Mode

Governance & Hygiene

As flag usage grows, discipline is key. Consider establishing formal governance rules:

- Flag Types : Categorize flags to clarify their purpose (e.g., release-toggle, experiment, ops-switch, kill-switch).
- Environments : Ensure your provider supports toggling flags independently across different environments (dev, staging, prod).
- Auditing : Your provider should have an audit log to track who created, toggled, or removed a flag.
- Lifecycle Metadata : Use tags or descriptions in your provider's UI to add context like owner: team-core-ui , jira: PROJ-123 , introduced: 2025-Q3 .

Treat feature flags as first-class citizens in your architecture. Without good hygiene, they become technical debt.

Compliance Notes

- Feature flags must be documented with purpose and cleanup plan
- All flag names must follow naming conventions
- Flag changes must be logged for audit purposes
- User targeting must respect privacy regulations
- Flag performance must be monitored
- Unused flags must be cleaned up regularly

Developer Checklist

- Does my new flag have a descriptive name and a cleanup plan?
- Have I provided a safe defaultValue for my isEnabled check?
- Is the flag check abstracted (e.g., in a variable) rather than being inline in the UI logic?
- Have I added the flag to the bootstrap config for offline testing?
- If the flag is long-term, have I discussed its purpose with the team?